Chapter 1. Introduction

In the history of software development, new approaches frequently bring discarded ideas back into the mainstream of common practice. Each time an idea is revisited, prior successes and failures become invaluable aides in improving the concept and making its implementation better, or at least more usable. Now I'm not saying that we keep reinventing the wheel; rather, we keep going back and improving the wheel. And doing so can often be the catalyst for new ideas and new technologies that were not possible with the old wheel.

We've seen centralized computing with mainframes and their associated terminals come back disguised as application servers and thin clients. We've seen the concept of P-Code return in the form of interpreted languages like Java and Visual Basic. The universe of software development seems to expand and contract like, well, the cosmic Universe. If you wait around long enough, you may just be able to use the work you're doing today at some time in the future.

The point is, really, that a good idea is a good idea, regardless of whether it's a new idea. Timeliness is what matters most. So it goes for the world of distributed computing. The concept isn't new, but it gets revisited constantly. Pervasive infrastructure and technologies like the Internet, web browsers, and their associated protocols have allowed us to go back and advance the state of distributed computing. The evolution's latest craze is web services.

Web services are basically server functions that have published the interface mechanisms needed to access their capabilities. They're being implemented in a wide variety of technologies, but have a very important thing in common: they are providers of computational services that can be accessed using a standardized protocol. For instance, you might find a stock quote service that can return current stock market pricing and trading information based on a company's stock symbol. This is a very specific function, and that's the essence of web services. They do not provide the breadth of capability found in application servers — they provide small, focused capabilities that are likely to prove useful when combined with other services. You can imagine an online trading application that makes use of web services ranging from stock quotes to trade execution to banking transactions. The vision of web services is that it will ultimately be possible to create complex applications on the fly — or at least, with minimal development time — by combining bits and pieces of data and services that are distributed across the Web. Sun's slogan used to be "The network is the computer," and that vision is certainly coming to fruition.

1.1 RPC and Message-Oriented Distributed Systems

Distributed systems exist, for the most part, as loosely coupled entities that communicate with each other to accomplish some task. One of the most common models used in distributed software is the remote procedure call (RPC). One reason for the popularity of RPC systems is that they closely resemble the function/method call syntax and semantics that we as programmers are so familiar with. Technologies like Java RMI, Microsoft's COM, and CORBA all use this kind of model. Of course, you have to jump through many hoops before making the ever-familiar method call to a remote system, but even with all that it still feels remarkably like making a local method call. Often, once the method call returns we don't care how it happened.[1] Much of the work in providing that abstraction to programmers at the API level is what makes up the majority of the distributed systems implementations.

Another popular model for distributed computing is message passing. Unlike the RPC model, messaging does not emulate the syntax of programming language function calls. Instead, structured data messages are passed between parties. These messages can serve to decouple the nodes of the distributed system somewhat, and message-based systems often prove to be more flexible than RPC-based systems. However, that flexibility can sometimes be just enough rope for programmers to hang themselves.

It seems that a reasonable, and powerful, compromise might be to combine these two models. Can you imagine a system that provides for the familiarity and ease of use of RPC-style invocations, along with some of the flexibility of message-type systems? It seems to me that it would require the definition of a data format that could describe itself, while at the same time conforming to a standard set of rules governing that very description. Hmmm . . .

1.2 Self-Describing Data

In programming parlance, the term *self-describing data* is itself self-describing. Put another way, if the question is, "What is self-describing data?" then the answer is, "Data that describes itself." Not a very useful definition. But that's the result of designing a flexible data format to be used by many, many people.

Let's look at a very simple example. Let's say we were designing a message-style distributed system for delivering stock quotes. We could design the response message format to be something like this:

- The first 10 characters contain the stock symbol, right-padded with spaces
- The next 10 characters represent the last price, right-padded with spaces
- The next 10 characters represent the volume, right-padded with spaces
- The next 20 characters represent the timestamp of the quote
- The next 10 characters represent the bid price
- The next 10 characters represent the ask price

You get the point. This is a fixed format message. It doesn't describe itself; rather it is described by the spec provided in the bullet list above. There's no flexibility here. And sometimes there's no need for any flexibility — it's not a one-size-fits-all scenario. But wouldn't you agree that there is some room for improvement? Let's consider the possibility that the values for last price, bid price, and ask price could be formatted in two different ways. The first format uses standard decimal notation, for example, 25.5. The other format uses fractions, so the same value would be encoded as 25 1/2. A self-describing format would have a provision for indicating which format is used for each of the price fields.

Now take this same concept and apply it to the overall structure of the message, as well as to its constituent parts. This gives you the flexibility to fully describe the contents of a message. For example, you could make some of the fields in the stock quote response optional. Maybe you've requested the quote after the market has closed, and maybe that renders the values for

[1] I'm not suggesting that you turn a blind eye to the fact that you're making calls to remote systems. Imagine, for instance, the ramifications of iterating over a remote array of ten thousand objects by using an array accessor method that goes out to the remote system for every array element. Not exactly efficient!

bid and ask prices useless. Then why return them at all? A self-describing format could specify its content, thereby having no need to return useless data.

In order for self-describing data to be truly useful, everyone has to use the same language for describing the data. I don't mean a programming language; I mean the language for expressing the description. What we need is a new way of describing and formatting these messages; a new grammar, so to speak. This new grammar would dictate the standard rules governing the format of these messages. In fact, this is where we really see the value of self-describing data: not so much to eliminate the problem of returning useless data, but to get everyone talking the same language. If I write a stock application that uses 11 characters to represent the stock symbol, it won't be able to talk to your stock server that uses 10 characters. But if we're exchanging self-describing data, the problem partially disappears: each piece of data says what it is in some standard way, so there's a much better chance that software coming from different people and organizations will be able to communicate.

1.3 XML

We've all been staring a form of self-describing data in the face every time we use a web browser. HTML is a good example of a standard data format that is quite flexible due to its provision for self-describing elements. For example, the color and font to be applied to a particular section of text are described right along with the text itself. This kind of self-describing data is commonly referred to as a *markup language*. The content is "marked-up" with instructions for its own presentation. This is very nice, and it obviously has gained an incredible level of industry acceptance. But HTML is not flexible enough to accommodate content that was not anticipated by its designers. That's not a knock on HTML; it's just the truth. HTML is not extensible.

The Extensible Markup Language (XML) is just what we're looking for. XML is a hierarchical, tag-based language much like HTML. The important difference for us is that it is fully extensible. It allows us to describe content that is specific to our own applications in a standard way, without the designers of the language having anticipated that content. For example, XML would allow me to create content to represent the stock quote response message from the previous section. It defines the rules that I must follow in order to accomplish that task, without dictating a specific format.

I'm not going to spend any time covering XML itself. If you're not familiar with XML, you may want to remedy that before you begin reading Chapter 2. Many books have been published on XML and related topics. A good starting point for general information is *XML in a Nutshell*, by Elliotte Rusty Harold and W. Scott Means (O'Reilly). For Java developers, *Java and XML* by Brett McLaughlin (O'Reilly) should be of particular interest.

1.4 API Specs Versus Wire-Level Specs

Java programmers are used to dealing with API-level specifications, where classes, interfaces, methods, and so on are clearly defined for the purpose of addressing a specific need. These specifications are designed to be independent of any specific implementation, focusing instead on the abstractions that must be implemented.

Consider the Java Message Service (JMS) specification. It fully describes the API that Java applications can use to access the features of message-oriented middleware (MOM) products.

The motivation for a standard API is simple: if MOM vendors adopt the API, it becomes that much easier for programmers to work with the various product offerings. In theory, you could swap one JMS implementation for another without impacting the rest of your code. In practice, it means that product vendors might be somewhat handcuffed, unable to provide alternative APIs that leverage features and capabilities of their own products without sacrificing compliance. Nevertheless, API specifications have been around a long time, and they do achieve most of what they're intended to do.

However, the API specification approach, by itself, leaves a gaping hole in an extremely important area of software development: interoperability. You can't develop an application using Vendor A's JMS-compliant API to communicate with Vendor B's JMS-compliant server. The specification deals with only the API, not the format of the data being communicated between the parties. This seems to suggest that interoperability is not as important as commonality of programming syntax. Yes, it's a trade-off, but it's not always the right one.

A wire-level specification, on the other hand, deals exclusively in the content and format of the data being transmitted between parties: the data that's "on the wire." Instead of concentrating on APIs, it devotes itself to the representation used for distributed computing interactions. So you can pretty much guarantee that if you work with implementations from more than one vendor, the APIs will not be the same. However, you have a decent chance of getting distributed systems that were built using products from multiple vendors to work together. If you're doing any work in the area of web services, the wire-level specification approach is your ally; the API specification approach won't get you very far.

1.5 Overview of SOAP

One of the more recent forays into the world of distributed computing resulted in a wire-level specification called the Simple Object Access Protocol, or SOAP. The protocol is relatively lightweight, is based on XML, and is designed for the exchange of information in a distributed computing environment. There is no concept of a central server in SOAP; all nodes can be considered equals, or even peers.

The protocol is made up of a number of distinct parts. The first is the *envelope*, used to describe the content of a message and some clues on how to go about processing it. The second part consists of the rules for encoding instances of custom data types. This is one of the most critical parts of SOAP: its extensibility. The last part describes the application of the envelope and the data encoding rules for representing RPC calls and responses, including the use of HTTP as the underlying transport.

RPC-style distributed computing is the most common Java usage of SOAP today, but it is not the only one. Any transport can be used to carry SOAP messages, even though HTTP is the only one described in the specification. There has been significant discussion of using SMTP, BEEP, JXTA, and other protocols for carrying SOAP messages.

1.5.1 Using SOAP with Java

SOAP differs from RMI, CORBA, and COM in that it concentrates on the content, effectively decoupling itself from both implementation and underlying transport. An interesting concept for a Java book, wouldn't you say? After all, implementation and transport are likely to be built using Java. Yet SOAP in no way addresses Java or any other implementation strategy.

The reality is that SOAP is an enabler, incapable of existing on its own beyond the abstraction of the specification. To benefit from SOAP, or any other protocol, there have to be real implementations. Java is a great technology for implementing SOAP, and for building web services and applications that use SOAP as the "on the wire" data format.

1.6 SOAP Implementations

As I write this book, there are dozens of SOAP implementations, and new ones emerge all the time. Some are implemented in Java, some aren't. Some are free, some aren't. And inevitably some are good, and some aren't. It would be impractical to do a side-by-side comparison of every available implementation or even to give equal coverage to them all. On the other hand, it wouldn't be wise to focus on a single implementation, since that would present a bias that I don't intend. A reasonable compromise, and the one I've elected to use, is to select two interesting Java SOAP implementations and use them both extensively throughout the book. This gives you an opportunity to see different APIs and programming strategies. In Chapter 9 and Chapter 11, I'll break this rule and look briefly at a couple of other important SOAP technologies.

1.6.1 Apache SOAP

The Apache Software Foundation has an ongoing project known as Apache SOAP. This is a Java implementation of the SOAP specification that can be hosted by servers that support Java servlets. The examples in this book are based on Apache SOAP Version 2.2, which is available at http://xml.apache.org/soap/index.html. Four very important factors led me to choose this implementation: it supports a good deal of the specification, it has a reasonably large user base, the source code is available, and it's free.

Although Apache SOAP can be hosted by a variety of server technologies, I've chosen Apache Tomcat (Version 3.3 and Version 4), available at http://jakarta.apache.org/tomcat/index.html. The reasons are not particularly tied to SOAP, but it does work well in Tomcat. The use of Tomcat has no real impact on the examples in the book, so you can feel free to select some other server technology if you like.

Apache SOAP was developed at a time when there were no standards for a SOAP API. The SOAP specification doesn't address language bindings, so the implementors were forced to come up with their own APIs. More recently, the Java community has been working on standards for SOAP-based messaging and RPC APIs, known as JAXM and JAX-RPC, respectively. We'll take a look at these APIs in Chapter 11; they will be implemented by Axis, which is the next-generation Apache SOAP implementation. In an ideal world, JAXM and JAX-RPC would have been completed in time for me to give them the coverage they deserve, since they will almost certainly replace the APIs developed for Apache SOAP 2.2. In reality, though, the standard APIs are just solidifying now, and should be in final form just in time to make me write a second edition earlier than I'd like. The bulk of this book will focus on technology that you can use to write production code now. Once you have the concepts down, moving to a different API will not be a challenge.

1.6.2 GLUE

GLUE is an implementation of SOAP developed by a company called The Mind Electric (http://www.themindelectric.com/). It's developed completely in Java, and can be hosted by a variety of servers that support servlets or can be run standalone using its own HTTP server. The GLUE examples in the book are based on GLUE Version 2, available at http://www.themindelectric.com/glue/index.html. I chose GLUE for four reasons as well: it uses a very programmer-friendly approach, its APIs are quite different from those found in Apache SOAP, it relies on the Web Services Description Language (WSDL), and it's free for most uses.[2]

The GLUE APIs are also proprietary. I'd expect that a future version of GLUE would adopt standards like JAXM and JAX-RPC if the user community demanded it, but that, of course, remains to be seen.

1.6.3 Others

In Chapter 9 we'll work a little with some other technologies. Microsoft's .NET is a major player in the area of SOAP-based web services, so we'll look at what it takes to write Java applications that use SOAP to communicate with .NET services.

The Apache Software Foundation is currently working on a next-generation SOAP implementation called Axis. Although it's still a bit early to cover Axis in any detail, there's no doubt that it will some day replace or subsume Apache SOAP Version 2.X. We'll take a peek at writing a simple Axis application using the current release. (And, as I've already said, Chapter 11 will look at the JAXM and JAX-RPC proposed standards, which Axis will eventually implement.)

1.7 The Approach

This book certainly does not cover every aspect of the SOAP technologies. My goal is to give you a good understanding of the major aspects of SOAP in the context of Java software development. You'll find that many of the examples are presented not only in Java source code, but also in the SOAP XML that is generated through the execution of the Java code. This will give you a sense of what the various APIs are actually accomplishing. Learning SOAP this way will allow you to go beyond the scope of this book with confidence, exploring the features and capabilities of the implementations I have covered as well those I have not.

1.7.1 No Security

One particular area is not covered in this book: security. How can you talk about a distributed computing technology without talking about security? The answer is actually quite simple: the SOAP specification does not deal with security. The current implementations rely on the security features of the hosting technology. Be it SSL, basic HTTP authentication, proxy authentication, or some other mechanism, all security is a function of the hosting technology and not part of SOAP itself. It's expected that either a future version of the SOAP spec or a separate SOAP Security spec will address that issue, but for now you'll have to rely on whatever your hosting technology supports.

[2] Please refer to the GLUE license agreement.

The current spec does, however, mention the use of SOAP headers in security schemes, and you will find that some SOAP implementations follow that lead. Nonetheless, the mechanisms are likely to be specific to each implementation until a standard is adopted.

1.8 Getting Started

If you plan to work with the examples, you'll need to install all of the necessary software components, including the JDK, the SOAP implementations described earlier, and a number of other supporting technologies. All of these packages are reasonably well documented, so you should have no trouble getting them installed properly.

The chapters in this book are arranged so that each one builds on concepts of the preceding chapters. I suggest that you follow along in order, but of course that's entirely up to you. If you are already comfortable with XML or (even better) with some aspects of web services, you may find that you can jump around a bit.